Heterogeneous Semantics and Unifying Theories

Jim Woodcock[1], Simon Foster[1], and Andrew Butterfield[2]

[1] Department of Computer Science, University of York, York YO10 5GH, UK.
{jim.woodcock,simon.foster}@york.ac.uk
[2] School of Computer Science and Statistics, Trinity College, University of Dublin, Dublin 2, Ireland
Andrew.Butterfield@scss.tcd.ie

Abstract. Model-driven development is being used increasingly in the development of modern computer-based systems. In the case of cyber-physical systems (including robotics and autonomous systems) no single modelling solution is adequate to cover all aspects of a system, such as discrete control, continuous dynamics, and communication networking. Instead, a heterogeneous modelling solution must be adopted. We propose a theory engineering technique involving Isabelle/HOL and Hoare & He's Unifying Theories of Programming. We illustrate this approach with mechanised theories for building a contractual theory of sequential programming, a theory of pointer-based programs, and the reactive theory underpinning CSP's process algebra. Galois connections provide the mechanism for linking these theories.

1 Introduction

Modern complex computer-based systems are often designed using model-based design techniques, checking models against specified requirements. Many diverse models may be needed to achieve this, encompassing software control, communication networking, and physical dynamics, all of which must contribute to the correct functioning of the system. This multi-paradigm approach involves different modelling languages and tools, including a wide range of analysis and simulation techniques. At present, there is neither a universal modelling language nor a universal tool for managing this diversity. Instead, modelling languages and tools must be used together cooperatively.

In this paper, we consider one approach to understanding heterogeneity in modelling and analysis, and how links can be made between different languages and their tools. We advocate mechanised theory engineering: the computer-supported development of definitions, axioms, and theorems encapsulating a particular concept. Theory engineering is the study of a concept in isolation, as well as exploring relationships between different concepts. The theory engineer builds coherent theory libraries, giving guidelines for building and adding new theories in order to support open semantic heterogeneity. We use Isabelle for this task, mechanising Unifying Theories of Programming (UTP), our chosen formalism for modelling language semantics [12].

Isabelle is an LCF-style interactive theorem prover: it has a special abstract type `thm` for theorems; the inference rules of the logical system are the constructors of the abstract type; it is implemented in a strongly typed high-level language. Logical correctness is enforced in the implementation language: everything of type `thm` has really been proved. The embedding in a full programming language allows the user to implement more sophisticated derived rules that decompose to the primitives without compromising soundness, allowing proof engineering at a much higher level than a simple proof checker.

Our embedding of UTP in Isabelle currently has three foundational theories: relations, designs, and reactive processes; further theories are under construction, including the hybrid relations. This allows us to build models of nondeterministic sequential programs, networks of reactive processes, and hybrid systems, including robotic and cyber-physical systems. These theories are accompanied by formalised and mechanised proofs of relevant properties. The theories themselves are structured as lattices linked by Galois connections, allowing for models to be translated between or embedded in different modelling paradigms.

In Sect. 2, we give an overview of UTP, and in Sect. 3, we give a detailed practical example of a UTP theory of separation logic. In Sect. 4, we consider the use of UTP in dealing with semantic heterogeneity by embedding the theory of designs in the theory of CSP processes. In Sect. 5, we draw some conclusions.

2 Unifying Theories of Programming

Unifying Theories of Programming (UTP) [17] is a long-term research agenda that records the relationship between different programming paradigms, both practical and theoretical. UTP has been widely used: Hoare & He formalise theories of sequential programming, with assertions; correct compilation; concurrent computation with reactive processes and communications; higher-order logic programming; and theories that link denotational, algebraic, and operational semantics [17]. More recent contributions include: angelic nondeterminism [23]; event-driven programs [34]; object orientation [25]; references [13]; probabilistic programs [2, 36]; real-time programs [16, 14]; timed reactive programs [26, 29]; and transaction processing [15]. Programming language semantics in UTP include: the hardware description languages Handel-C [21] and Verilog [35]; the multi-paradigm languages *Circus* [20] and CML [30]; Safety-Critical Java [7]; and Simulink [6]. A wide variety of programming theories have been formalised in UTP, including theories of confidentiality [1], testing [4], and undefinedness [31]. UTP has been embedded in a variety of theorem provers, notably in ProofPower Z and Isabelle [19, 32, 11]. This allows a theory engineer to mechanically construct UTP theories, experiment, prove properties, and eventually deploy them for use in program verification. In this paper, we focus on Isabelle/UTP [11].

UTP gives three principal ways to study the relationships between different programming paradigms. UTP classifies languages according to their computational model. Common concepts are identified and variations treated separately. A different categorisation is by level of abstraction within a particular paradigm.

This might range from platform-specific implementation technology at the bottom, and very high-level description of overall requirements at the top end. Between these, there are descriptions of components and their architectures. Each level has contractual interfaces, and UTP gives ways of mapping between these levels based on a formal notion of refinement that provides guarantees of correctness all the way from requirements to code. The final classification is by the method chosen to present a language definition. Three widely used scientific methods are: *denotational*, *algebraic*, and *operational*. As Hoare & He point out [17], a comprehensive account of a programming theory needs all three kinds of presentation, and the UTP technique allows us to study differences and mutual embeddings, and to derive each from the others.

The UTP research agenda has as its ultimate goal to cover all the interesting paradigms of computing, including hardware [21, 37], hardware/software codesign [3] and component-based systems [33]. But it also presents an opportunity when constructing new languages, especially ones with heterogeneous paradigms and techniques. UTP uses an alphabetised version of Tarski's relational calculus, presented in a predicative style. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions. *The alphabet* is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. *The signature* gives the rules for the syntax for denoting objects of the theory. *Healthiness conditions* identify properties that characterise the predicates of the theory. Each healthiness condition embodies an important fact about the computational model for the programs being studied.

Example 1 (Nondeterministic sequential programming language). The signature for designs consists of assignment ($x := e$), sequential composition ($P \ ; \ Q$), conditional choice ($P \lhd b \rhd Q$), nondeterministic choice ($P \sqcap Q$), and recursion ($P = F(P)$). The only observations that can be made are of the program variables. There are no healthiness conditions for this simple programming language. The program operators are given the following meanings:

Command	**Semantics**	**Alphabet**
$x := e$	$(x' = e) \wedge (v' = v)$	$\{x, v, x', v'\}$
$P \ ; \ Q$	$\exists\, v_0 \bullet P[v_0/v'] \wedge Q[v_0/v]$	$in\alpha P \cup out\alpha Q$
$P \sqcap Q$	$P \vee Q$	$\alpha P \cup \alpha Q$
$P \lhd b \rhd Q$	$(P \wedge b) \vee (Q \wedge \neg\, b)$	$\alpha P = \alpha Q \supseteq \alpha b$
$P = F(P)$	νF (the strongest fixed point of F)	αP

Example 2 (Hoare logic). Hoare logic is a set of axioms and inference rules for reasoning formally about the correctness of programs. The central feature of Hoare logic is the Hoare triple, which describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form $\{\, p \,\}\ Q\ \{\, r \,\}$, where p and r are predicates on the program state (the precondition and the postcondition respectively) and Q is a command build from the signature

of our programming language. Standard Hoare logic provides a way of reasoning about partial correctness; termination needs to be proved separately. The next definition defines the denotation of a Hoare triple.

Definition 3 (Hoare triple [17]).

$$\{\, p \,\}\ Q\ \{\, r \,\} \ \widehat{=}\ [\, Q \Rightarrow (p \Rightarrow r') \,] \qquad = \qquad (p \Rightarrow r') \sqsubseteq Q$$

The axioms and inference rules are proved as theorems in UTP, providing the first link in this paper between different semantics: axiomatic and denotational.

Definition 4 (Hoare logic).

L1 ***if*** $\{\, p \,\}\ Q\ \{\, r \,\}$ ***and*** $\{\, p \,\}\ Q\ \{\, s \,\}$ ***then*** $\{\, p \,\}\ Q\ \{\, r \wedge s \,\}$
L2 ***if*** $\{\, p \,\}\ Q\ \{\, r \,\}$ ***and*** $\{\, q \,\}\ Q\ \{\, r \,\}$ ***then*** $\{\, p \vee q \,\}\ Q\ \{\, r \,\}$
L3 ***if*** $\{\, p \,\}\ Q\ \{\, r \,\}$ ***then*** $\{\, p \wedge q \,\}\ Q\ \{\, r \vee s \,\}$

L4 $\{\, r[e/x] \,\}\ x := e\ \{\, r \,\}$
L5 ***if*** $\{\, p \wedge b \,\}\ Q_1\ \{\, r \,\}$ ***and*** $\{\, p \wedge \neg\, b \,\}\ Q_2\ \{\, r \,\}$
then $\{\, p \,\}\ Q_1 \lhd b \rhd Q_2\ \{\, r \,\}$
L6 ***if*** $\{\, p \,\}\ Q_1\ \{\, s \,\}$ ***and*** $\{\, s \,\}\ Q_2\ \{\, r \,\}$ ***then*** $\{\, p \,\}\ Q_1 \,;\, Q_2\ \{\, r \,\}$

L7 ***if*** $\{\, p \,\}\ Q_1\ \{\, r \,\}$ ***and*** $\{\, p \,\}\ Q_2\ \{\, r \,\}$ ***then*** $\{\, p \,\}\ Q_1 \sqcap Q_2\ \{\, r \,\}$
L8 ***if*** $\{\, b \wedge c \,\}\ Q\ \{\, c \,\}$
then $\{\, c \,\}\ \nu X \bullet (Q \,;\, X) \lhd b \rhd \text{I\!I}\ \{\, \neg\, b \wedge c \,\}$
L9 $\{\, \mathit{false} \,\}\ Q\ \{\, r \,\}$ ***and*** $\{\, p \,\}\ Q\ \{\, \mathit{true} \,\}$
and $\{\, p \,\}$ ***false*** $\{\, \mathit{false} \,\}$ ***and*** $\{\, p \,\}\ \text{I\!I}\ \{\, p \,\}$

Example 5 (Designs). The relational theory is adequate for describing partial correctness; termination requires a more expressive semantics. The signature of the programming language introduced in Example 1 is extended with the syntax of a design, $P \vdash Q$, with precondition P and postcondition Q [28]. The alphabet contains two boolean variables: ok, which is the observation that the program has started; and ok', which is the observation that the program has terminated. Each of these variables has a corresponding healthiness condition.

$$\mathbf{H1}(P) \ \widehat{=}\ ok \Rightarrow P$$
$$\mathbf{H2}(P) \ \widehat{=}\ P \,;\, J \quad \text{where } J = (ok \Rightarrow ok') \wedge (v' = v), \alpha P = \{v, v', ok, ok'\}$$

H1 ensures that no observation may be made of P's behaviour until after the program has started. ***H2*** says that P is monotonic with respect to the ok' variable: one of the behaviours of an aborting program is unexpectedly to terminate. Both healthiness conditions are monotonic idenpotents. We define $\mathbf{H} = \mathbf{H1} \circ \mathbf{H2}$. Finally, we define the design $P \vdash Q$ as the single relation $ok \wedge P \Rightarrow ok' \wedge Q$.

In advance of our discussion of separation logic, the following example shows the use of the assignment axiom from Hoare logic.

Example 6 (Programming with assertions). Consider the following outline of a Java class that keeps track of a bank account where overdrafts are not permitted:

```
class BankAccount {
  private int balance;
  { invariant : balance >= 0 }
  ...
  deposit(int x){
    { precondition : x > 0 }
    // is the invariant preserved?
    // is balance >= 0?
    ...
  }
}
```

We need to prove that `deposit` preserves the class invariant $balance \geq 0$. The assignment axiom tells us that the precondition for this is that $balance + x \geq 0$. This weaker precondition follows from a stronger one that involves the class invariant before executing `deposit` and the precondition stated for the method: $balance \geq 0 \wedge x > 0$ (**L3** in Def. 4). Both are valid assumptions.

3 Example Theory: Separation Logic

In this section, we present our basic theory for separation logic [24]. We start with a motivating example.

Example 7 (Hoare logic is unsound wrt aliasing). Consider the assignment axiom from Hoare Logic: $\{ [E/x]P \}$ `x := e` $\{ P \}$,which represents the fact that the value of a variable `x` after executing an assignment command `x := E` equals the value of the expression `E` in the state before executing it. Formally, if P is to be true after the assignment, then the statement obtained by substituting E for x in P must be true before executing it. Now consider the following program:

```
x := (new Cell(3,nil));80
y := x;
y.head := 4
```

Perversely, let's prove that the program makes the variables `x` and `y` distinct. Here, we need the assignment axiom and the proof rules for sequential composition and consequence (read the proof outline from the bottom to the top):

$\{true\}$
$\{4 > 3\}$
$\{4 > (new\ Cell(3, nil)).head\}$
`x := new Cell(3, nil)`
$\{4 > x.head\}$
`y:=x`

$\{4 > x.head\}$

```
y.head := 4
```

$\{y.head > x.head\}$

So, the program always has the postcondition $y.head > x.head$, even though `x` and `y` point to the same `Cell` object! We can tell that something is wrong here, since this doesn't match the expected semantics. It turns out that it's the assignment axiom that's at fault: it's unsound in the presence of aliasing.

Example 7 illustrates a classical problem in Computer Science: *the aliasing problem*. This comes about from using standard programming features: call-by-reference parameters and pointer variables. To overcome the soundness problem, we need more discrimination in our semantic model and inference rules. The frame problem is familiar elsewhere. In AI, it is the challenge of representing the effects of action in logic without having to represent explicitly a large number of intuitively obvious non-effects. More generally, it is about modular reasoning.

Separation logic is one of a number of approaches that solve this problem of unsoundness. It was developed by Reynolds and O'Hearn, based on some early work by Burstall. It helps a programmer to reason about programs that manipulate pointer data structures. More generally, it helps with modular reasoning about ownership of resources and virtual separation between concurrent processes. Our theory of separation logic (`utp_seplog`) in mechanised in Isabelle/UTP and builds upon the theories of `utp_designs` and `utp_invariants`.

We introduce three uninterpreted datatypes: *Var*, the set of program variables names (ranged over by x and y); *Loc*, the set of heap addresses (ranged over by l); and *Val*, the set of values manipulated by a program. As well as program variables, the following observations made be made of a program.

1. $fp : \mathbb{F}\, Loc$ The footprint of the program, a finite set of heap addresses.
2. $st :: Var \nrightarrow Val \cup Loc$ The store: the denotations for variables, a finite function from variable names to values or heap addresses.
3. $hp :: Loc \nrightarrow Val \cup Loc$ The heap: the contents of the heap addresses, a finite function from heap addresses to values or further heap addresses.

3.1 Healthiness conditions

Predicates in the theory of separation logic satisfy four healthiness conditions. (i) Nothing changes outside the footprint. (ii) The footprint contains only heap addresses. (iii) A program is independent of the heap outside its footprint. (iv) Every address used in the store or on the heap is itself a heap address (no dangling pointers). These conditions are formalised in the following definition.

Definition 8.

$$\mathbf{SL1}(P) \mathrel{\widehat{=}} \mathbf{OIH}((fp' \lhd hp' = fp' \lhd hp))(P)$$
$$\mathbf{SL2}(P) \mathrel{\widehat{=}} \mathbf{OSH}(fp \subseteq \mathrm{dom}\, hp)(P)$$

$$\textbf{SL3}(P) \;\widehat{=}\; \prod hp_0 \mid hp_0 \subseteq hp \wedge \operatorname{dom} hp_0 \cap fp' = \varnothing \bullet hp :=_D hp \setminus hp_0 \,;\, P \,;\, hp :=_D hp \cup hp_0$$

$$\textbf{SL4}(P) \;\widehat{=}\; \textbf{OSH}(\forall\, l \mid l \in \operatorname{ran}(st) \cup \operatorname{ran} hp \bullet l \in \operatorname{dom} hp)(P)$$

where $\textbf{OIH}(I)(P) = P \wedge (ok \wedge \neg P^f \Rightarrow I)$

$\textbf{OSH}(q)(P) = P \wedge (ok \wedge \neg P^f \wedge q \Rightarrow q')$

OIH *imposes an operation invariant and* **OSH** *output-state healthiness [7].*

Theorem 9. ***SL1–4*** *are monotonic idempotents that mutually commute.*

The next theorem is important in reasoning about heap predicates. First, an enabling lemma.

Lemma 10 (Contraction). *For* $\operatorname{dom} hp \cap \operatorname{dom} hp_0$ *and that* $\operatorname{dom} hp_0 \cap fp' = \varnothing$.

$$hp :=_D hp \setminus hp_0 \,;\, P \,;\, = \;\; hp :=_D hp \cup hp_0$$

New heap addresses added by P *lie in* $fp' \setminus fp$, *and the* hp_0*'s contribution is*

$$(fp' \setminus fp) \cap \operatorname{dom} hp_0$$

which is empty, since $\operatorname{dom} hp_0 \cap fp'$ *is empty by assumption. Disposed heap addresses lie in the set* $fp \setminus fp'$. *So, the new heap addresses (*$\operatorname{dom} hp'$*) are*

$$((\operatorname{dom} hp) \setminus (fp \setminus fp')) \cup (fp' \setminus fp)$$

which is clearly disjoint from $\operatorname{dom} hp_0$, *since* $(\operatorname{dom} hp) \setminus (fp \setminus fp') \subseteq \operatorname{dom} hp$, *which is disjoint from* $\operatorname{dom} hp_0$ *by assumption.* ***SL3****'s following assignment is*

$$\begin{array}{l}
(P_1 \vdash P2) \,;\, hp :=_D hp \cup hp_0 \\
= \{ \textit{definition: design assignment} \} \\
(P_1 \vdash P2) \,;\, (\textbf{true} \vdash hp := hp \cup hp_0) \\
= \{ \textit{design composition, simplification} \} \\
(P_1 \vdash P2 \,;\, hp := hp \cup hp_0) \\
= \{ \textit{relational assignment} \} \\
(P_1 \vdash P2 \,;\, (hp' = hp \cup hp_0) \wedge (st' = st) \wedge (fp' = fp)) \\
= \{ \textit{from above, } \operatorname{dom} hp \cap \operatorname{dom} hp_0 = \varnothing \} \\
(P_1 \vdash P2 \,;\, (hp = hp' \setminus hp_0) \wedge (st' = st) \wedge (fp' = fp)) \\
= \{ \textit{relational calculus} \} \\
(P_1 \vdash P2[hp' \setminus hp_0 / hp']) \\
= \{ \textit{assumption: } hp' \textit{ not free in } P_1\textit{, substitution shorthand} \} \\
P^{hp' \setminus hp_0}
\end{array}$$

Now consider the leading assignment too:

$$\begin{array}{l}
hp :=_D hp \setminus hp_0 \,;\, P^{hp' \setminus hp_0} \\
= \{ \textit{design calculus: leading assignment} \} \\
P^{hp' \setminus hp_0}_{hp \setminus hp_0}
\end{array}$$

Theorem 11 (Contraction). *If P is **SL3**-healthy, then for all hp_0, such that $hp_0 \subseteq hp \wedge \operatorname{dom} hp_0 \cap fp' = \varnothing$*

$$P \sqsubseteq P^{hp' \setminus hp_0}_{hp \setminus hp_0}$$

Proof. ***SL3***(P) *is a greatest lower-bound and Lemma 10.*

3.2 Signature

We add five atomic heap assignment commands to the signature of the nondeterministic sequential programming language introduced in Example 1.

$$\mathcal{C} \;::=\; x := y \;\mid\; [x] := v \;\mid\; [x] := y \;\mid\; x := [y] \;\mid\; x := \mathsf{ref}\ y$$

The following definitions explain the semantics of each of these assignments.

Definition 12 (vv-assign). *The variable-variable assignment $x :=_s y$ assigns to the variable x the denotation of y, namely $st(y)$, which must be well defined.*

$$\begin{array}{|l} \_ :=_s \_ : \mathit{Var} \leftrightarrow \mathit{Var} \\ \hline x :=_s y \;=\; y \in \operatorname{dom}(st) \vdash st := st \cup \{x \mapsto st(y)\} \end{array}$$

Definition 13 (pc-assign). *The pointer-constant assignment $[x]_c :=_s v$ updates the heap location pointed to by the denotation of x, namely $st(x)$, to hold the value v. This command's footprint is exactly the location $st(x)$. The denotation $st(x)$ must be well defined and its valuation $st(x)$ must be current.*

$$\begin{array}{|l} [\_]_c :=_s \_ : \mathit{Var} \leftrightarrow \mathit{Val} \cup \mathit{Loc} \\ \hline [x]_c :=_s v \;=\; \left(\begin{array}{c} x \in \operatorname{dom}(st) \wedge st(x) \in \operatorname{dom} hp \\ \vdash \\ hp, fp := hp \cup \{st(x) \mapsto v\}, fp \cup \{st(x)\} \end{array}\right) \end{array}$$

Definition 14 (pv-assign). *The pointer-variable assignment $[x] :=_s y$ updates the heap location pointed to by the denotation of x, namely $st(x)$, to hold the value denoted by the variable y, namely $st(y)$. The footprint is exactly $st(x)$. Both $st(x)$ and $st(y)$ must be well defined and $st(x)$ must be a heap address.*

$$\begin{array}{|l} [\_] :=_s \_ :: \mathit{Var} \leftrightarrow \mathit{Var} \\ \hline [x] :=_s y \;=\; \left(\begin{array}{c} x \in \operatorname{dom}(st) \wedge y \in \operatorname{dom}(st) \wedge st(x) \in \operatorname{dom} hp \\ \vdash \\ hp, fp := hp \cup \{st(x) \mapsto st(y)\}, fp \cup \{st(x)\} \end{array}\right) \end{array}$$

Definition 15 (vp-assign). *The variable-pointer assignment $x :=_s [y]$ assigns to the variable x the denotation of the location of y. The footprint of this command is exactly $st(y)$, which must be well defined and be a heap address.*

$$\begin{array}{|l} \_ :=_s [\_] :: \mathit{Var} \leftrightarrow \mathit{Var} \\ \hline x :=_s [y] \;=\; \left(\begin{array}{c} y \in \operatorname{dom}(st) \wedge st(y) \in \operatorname{dom} hp \\ \vdash \\ st, fp := st \cup \{x \mapsto hp(st(y))\}, fp \cup \{st(y)\} \end{array}\right) \end{array}$$

Definition 16 (vr-assign). *The variable-reference assignment* $x :=_s \textbf{\textit{ref}}\ y$ *assigns to* x *a fresh reference on the heap pointing to the denotation of* y*. Freshness means that the new reference is not on the current heap. The denotation* $st(y)$ *must be well defined. The footprint is exactly the new reference.*

$$\_ :=_s ref\_ :: Var \leftrightarrow Var$$

$$x :=_s \textbf{\textit{ref}}\ y \ = \ \exists\, l \bullet \left(\begin{array}{c} y \in \mathrm{dom}(st) \\ \vdash \\ l \notin \mathrm{dom}\, hp \\ \left(\begin{array}{c} st \\ hp \\ fp \end{array} \right) := \left(\begin{array}{l} st \cup \{x \mapsto l\} \\ hp \cup \{l \mapsto st(y)\} \\ fp \cup \{l\} \end{array} \right) \end{array} \right)$$

The vv-assignment command is healthy.

Theorem 17 $(x :=_s y\_is\_\textbf{\textit{SL}}\_healthy)$**.**

$$(x :=_s y)\ is\ \textbf{\textit{SL1}} \circ \textbf{\textit{SL2}} \circ \textbf{\textit{SL3}} \circ \textbf{\textit{SL4}}$$

We prove the third part of the theorem.

Lemma 18 $(x :=_s y\_is\_\textbf{\textit{SL3}})$**.**

$$(x :=_s y)\ is\ \textbf{\textit{SL3}}$$

Proof.

$$\begin{array}{l} \textbf{\textit{SL3}}(x :=_s y) \\ = \{\, x :=_s y\_def,\ \textbf{\textit{SL3}}\_def \,\} \\ \bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \mathrm{dom}\, hp_0 \cap fp' = \varnothing \bullet \\ \quad hp :=_D hp \setminus hp_0 \,;\, (y \in \mathrm{dom}(st) \vdash st := st \cup \{x \mapsto st(y)\}) \,;\, hp :=_D hp \cup hp_0 \\ = \{\, \textit{leading, following assignment} \,\} \\ \bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \mathrm{dom}\, hp_0 \cap fp' = \varnothing \bullet \\ \quad (y \in \mathrm{dom}(st) \vdash (st := st \cup \{x \mapsto st(y)\})[hp \setminus hp_0/hp] \,;\, hp := hp \cup hp_0) \\ = \{\, \textit{substitution} \,\} \\ \bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \mathrm{dom}\, hp_0 \cap fp' = \varnothing \bullet \\ \quad (y \in \mathrm{dom}(st) \vdash st, hp := st \cup \{x \mapsto st(y)\}, hp \setminus hp_0 \,;\, hp := hp \cup hp_0) \\ = \{\, \textit{assignment composition: } x := e \,;\, x := f(x) \ = \ x := f(e) \,\} \\ \bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \mathrm{dom}\, hp_0 \cap fp' = \varnothing \bullet \\ \quad (y \in \mathrm{dom}(st) \vdash st, hp := st \cup \{x \mapsto st(y)\}, (hp \setminus hp_0) \cup hp_0) \\ = \{\, \textit{lemma: } hp_0 \subseteq hp \Rightarrow (hp \setminus hp_0) \cup hp_0 = hp \,\} \\ \bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \mathrm{dom}\, hp_0 \cap fp' = \varnothing \bullet (y \in \mathrm{dom}(st) \vdash st := st \cup \{x \mapsto st(y)\}) \\ = \{\, \textit{lemma: } (\bigsqcap x \mid P \bullet Q) = Q \textit{, providing } \exists\, x \bullet P \textit{ and } x \textit{ not free in } Q \,\} \\ y \in \mathrm{dom}(st) \vdash st := st \cup \{x \mapsto st(y)\} \end{array}$$

$= \{\ x :=_s\ y\_def\ \}$

$x :=_s\ y$

The vr-assignment is healthy.

Theorem 19 ($x :=_s$ ***ref*** $y\_is\_$***SL***$\_healthy$)**.**

$(x :=_s$ ***ref*** $y)$ *is* ***SL1*** $\circ$ ***SL2*** $\circ$ ***SL3*** $\circ$ ***SL4***

Again, we prove the third part of the theorem.

Lemma 20. *Proof.*

SL3$(x :=_s$ ***ref*** $y)$

$= \{\ vr\_assign\ \}$

$$\bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \operatorname{dom} hp_0 \cap fp' = \varnothing \bullet hp :=_D hp \setminus hp_0 \,;\, \exists\, l \bullet \left(\begin{array}{c} y \in \operatorname{dom}(st) \\ \vdash \\ l \notin \operatorname{dom} hp \\ \begin{pmatrix} st \\ hp \\ fp \end{pmatrix} := \begin{pmatrix} st \cup \{x \mapsto l\} \\ hp \cup \{l \mapsto st(y)\} \\ fp \cup \{l\} \end{pmatrix} \end{array} \right) \,;\, hp :=_D hp \cup hp_0$$

$$= \left\{ \begin{array}{l} \textit{lemma: } x :=_D e \,;\, (q_1 \vdash Q_2) \;=\; (q_1[e/x] \vdash Q_2[e/x]), \\ \textit{lemma: } (p_1 \vdash P2) \,;\, x :=_D f \;=\; (p_1 \vdash P_2 \,;\, x := f) \end{array} \right\}$$

$$\bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \operatorname{dom} hp_0 \cap fp' = \varnothing \bullet \exists\, l \bullet \left(\begin{array}{c} y \in \operatorname{dom}(st) \\ \vdash \\ l \notin \operatorname{dom}(hp \setminus hp_0) \\ \begin{pmatrix} st \\ hp \\ fp \end{pmatrix} := \begin{pmatrix} st \cup \{x \mapsto l\} \\ (hp \setminus hp_0) \cup \{l \mapsto st(y)\} \\ fp \cup \{l\} \end{pmatrix} \,;\, hp := hp \cup hp_0 \end{array} \right)$$

$= \{\ \textit{assignment composition}\ \}$

$$\bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \operatorname{dom} hp_0 \cap fp' = \varnothing \bullet \exists\, l \bullet \left(\begin{array}{c} y \in \operatorname{dom}(st) \\ \vdash \\ l \notin \operatorname{dom}(hp \setminus hp_0) \\ \begin{pmatrix} st \\ hp \\ fp \end{pmatrix} := \begin{pmatrix} st \cup \{x \mapsto l\} \\ (hp \setminus hp_0) \cup \{l \mapsto st(y)\} \cup hp_0 \\ fp \cup \{l\} \end{pmatrix} \end{array} \right)$$

$= \{\ \textit{lemma: } hp_0 \subseteq hp \Rightarrow (hp \setminus hp_0) \cup hp_0 = hp \textit{ and commutativity of } \cup\ \}$

$$\bigsqcap hp_0 \mid hp_0 \subseteq hp \wedge \operatorname{dom} hp_0 \cap fp' = \varnothing \bullet \exists\, l \bullet \left(\begin{array}{c} y \in \operatorname{dom}(st) \\ \vdash \\ l \notin \operatorname{dom}(hp \setminus hp_0) \\ (st, hp, fp) := (st \cup \{x \mapsto l\}, hp \cup \{l \mapsto st(y)\}, fp \cup \{l\}) \end{array} \right)$$

$$
\begin{array}{l}
= \{\, l \in fp' \land \operatorname{dom} hp_0 \cap fp' = \varnothing \Rightarrow l \notin \operatorname{dom} hp_0 \,\} \\
\bigsqcap hp_0 \mid hp_0 \subseteq hp \land \operatorname{dom} hp_0 \cap fp' = \varnothing \bullet \\
\qquad \exists\, l \bullet \left(\begin{array}{c} y \in \operatorname{dom}(st) \\ \vdash \\ l \notin \operatorname{dom} hp \\ (st, hp, fp) := (st \cup \{x \mapsto l\}, hp \cup \{l \mapsto st(y)\}, fp \cup \{l\}) \end{array} \right) \\
= \{\, \textit{lemma: } (\bigsqcap x \mid P \bullet Q) = Q \textit{, providing } \exists\, x \bullet P \textit{ and } x \textit{ not free in } Q \,\} \\
\exists\, l \bullet \left(\begin{array}{c} y \in \operatorname{dom}(st) \\ \vdash \\ l \notin \operatorname{dom} hp \land (st, hp, fp) := (st \cup \{x \mapsto l\}, hp \cup \{l \mapsto st(y)\}, fp \cup \{l\}) \end{array} \right) \\
= \{\, vr\_assign \,\} \\
x :=_s \textbf{\textit{ref}}\ y
\end{array}
$$

SL-healthy predicates support sound modular reasoning about pointer programs. Next, we describe the essential part of separation logic that achieves this.

3.3 Separating conjunction

Two disjoint heaplets can be joined compatibly:

Definition 21 (Compatible join).

$$st \circledast (s_1, s_2) \ \widehat{=}\ \operatorname{dom} s_1 \cap \operatorname{dom} s_2 = \varnothing \land st = s_1 \cup s_2$$

The binary operator $*$ (pronounced "star" or "separating conjunction") asserts that the heap can be split into two disjoint parts where its two arguments hold.

Definition 22 (Separating conjunction).

$$p * q \ \widehat{=}\ \exists\, h_1, h_2 \bullet hp \circledast (h_1, h_2) \land p_{h_1} \land q_{h_2}$$

In order to be able to give a meaning to exceptional faulting states, our theory of separation logic will be a subset embedding of our theory of designs. This means that we must revise our notion of Hoare logic for total correctness.

Definition 23 (Hoare triple revisited).

$$\{\, p \,\}\ Q\ \{\, r \,\} \ =\ (p \Rightarrow r') \sqsubseteq Q \qquad\qquad [\, p \Rightarrow \mathsf{fv}(Q) \subseteq \operatorname{dom} st \,]$$

The proviso formulation is due to Reynolds: p ensures Q cannot abort due to dangling pointers. Essentially

$$[\, p \Rightarrow \mathsf{fv}(Q) \subseteq \operatorname{dom} st \land (Q \Rightarrow r') \,]$$

Now we augment Hoare logic with separation logic's Frame Rule. This states that if Q can execute safely in a local state satisfying p, then it can also execute in any larger state satisfying $p * s$. This idea will be familiar from the semantics that we have presented so far. The footprint for an ***SL***=healthy predicate P is an observation that describes a sufficiently large heap for P to execute satisfactorily. The minimal footprint adds necessity, but any larger heap will do. In what follows, we use the following shorthands $p_e = p[e/hp]$ $Q^f_e = Q[e, f/hp, hp']$.

Theorem 24 (Frame Rule). *Suppose that Q is **SL** and that Q's use of the store is no wider that that of the precondition p. This inference rule is valid:*

$$\frac{\{\, p \,\}\ Q\ \{\, r \,\}}{\{\, p * s \,\}\ Q\ \{\, r * s \,\}}\ [\, \mathsf{use}(Q) \cap \mathsf{use}(s) = \varnothing \,]$$

Proof.

$\{\, p \,\}\ Q\ \{\, r \,\} \Rightarrow \{\, p * s \,\}\ Q\ \{\, r * s \,\}$
$= \{$ *Def. 3 (Hoare triple)* $\}$
$\{\, p \,\}\ Q\ \{\, r \,\} \Rightarrow [\, p * s \wedge Q \Rightarrow (r * s)_{hp'} \,]$
$\Leftarrow \{$ *predicate calculus: $\forall$-I, arbitrary hp and hp'* $\}$
$\{\, p \,\}\ Q\ \{\, r \,\} \wedge (p * s) \wedge Q \Rightarrow (r * s)_{hp'}$
$= \{$ *Def. 22 (separating conjunction)* $\}$
$\{\, p \,\}\ Q\ \{\, r \,\} \wedge (\exists\, hp_1, hp_2 \bullet hp \circledast (hp_1, hp_2) \wedge p_{hp_1} \wedge s_{hp_2}) \wedge Q \Rightarrow (r * s)_{hp'}$
$\Leftarrow \{$ *predicate calculus: $\exists$-E, arbitrary hp_1 and hp_2* $\}$
$\{\, p \,\}\ Q\ \{\, r \,\} \wedge hp \circledast (hp_1, hp_2) \wedge p_{hp_1} \wedge s_{hp_2} \wedge Q \Rightarrow (r * s)_{hp'}$
$\Leftarrow \{$ *Q is **SL3**, Theorem 11 (Contraction)* $\}$
$\{\, p \,\}\ Q\ \{\, r \,\} \wedge hp \circledast (hp_1, hp_2) \wedge p_{hp_1} \wedge s_{hp_2} \wedge Q^{hp' \setminus hp_2}_{hp \setminus hp_2} \Rightarrow (r * s)_{hp'}$
$= \{$ *Def. 3 (Hoare triple)* $\}$
$[\, p \wedge Q \Rightarrow r_{hp'} \,] \wedge hp \circledast (hp_1, hp_2) \wedge p_{hp_1} \wedge s_{hp_2} \wedge Q^{hp' \setminus hp_2}_{hp \setminus hp_2} \Rightarrow (r * s)_{hp'}$
$\Leftarrow \{$ *predicate calculus: $\forall$-E, $hp \setminus hp_2, hp' \setminus hp_2 / hp, hp'$* $\}$
$(p_{hp \setminus hp_2} \wedge Q^{hp' \setminus hp_2}_{hp \setminus hp_2} \Rightarrow r_{hp' \setminus hp_2}) \wedge p_{hp_1} \wedge s_{hp_2} \wedge hp \circledast (hp_1, hp_2) \wedge Q^{hp' \setminus hp_2}_{hp \setminus hp_2} \Rightarrow (r * s)_{hp'}$
$\Leftarrow \{$ *lemma: $hp \circledast (hp_1, hp_2) \Rightarrow hp_1 = hp \setminus hp_2$* $\}$
$\wedge\, (p_{hp \setminus hp_2} \wedge Q^{hp' \setminus hp_2}_{hp \setminus hp_2} \Rightarrow r_{hp' \setminus hp_2}) \wedge p_{hp \setminus hp_2} \wedge s_{hp_2} \wedge Q^{hp' \setminus hp_2}_{hp \setminus hp_2} \Rightarrow (r * s)_{hp'}$
$\Leftarrow \{$ *propositional calculus: $\wedge$-E* $\}$
$r_{hp' \setminus hp_2} \wedge s_{hp_2} \Rightarrow (r * s)_{hp'}$
$= \{$ *Def. 22 (separating conjunction)* $\}$
$r_{hp' \setminus hp_2} \wedge s_{hp_2} \Rightarrow \exists\, hp'_1, hp'_2 \bullet hp' \circledast (hp'_1, hp'_2) \wedge r_{hp'_1} \wedge s_{hp'_2}$
$\Leftarrow \{$ *predicate calculus: $\exists$-I, $(hp' \setminus hp_2), hp2 / hp'_1, hp'_2$* $\}$
$r_{hp' \setminus hp_2} \wedge s_{hp_2} \Rightarrow hp' \circledast (hp' \setminus hp_2, hp_2) \wedge r_{hp' \setminus hp_2} \wedge s_{hp_2}$
$= \{$ *lemma: $hp' \circledast (hp' \setminus hp_2, hp_2)$* $\}$
$r_{hp' \setminus hp_2} \wedge s_{hp_2} \Rightarrow r_{hp' \setminus hp_2} \wedge s_{hp_2}$
$= \{$ *propositional calculus: tautology* $\}$
$true$

This proof is the longest in this paper. The key step is Theorem 11 (Contraction).

4 Heterogeneous Semantics

In this section, we describe the mechanism that we use to connect heterogeneous semantics coherently: the Galois connection.

Definition 25 (Galois connection). *(L, R) is a Galois connection between lattices S and T iff the following three conditions hold:*

1. *L and R are both monotonic.*
2. *$L \circ R \sqsupseteq id_T$ (strengthening).*
3. *$id_S \sqsupseteq R \circ L$ (weakening).*

If $L \circ R = id_T$ (or L is surjective or R is injective), then (L, R) is a retract. If $R \circ L = id_S$ (or R is surjective or L is injective), then (L, R) is a coretract.

To illustrate the use of Galois connections in heterogeneous semantics, consider the UTP theory of CSP processes [8]. We start with the theory of reactive processes.

Definition 26 (Reactive processes). *A reactive process has the following observations: (i) A trace tr of events that have occurred up to the moment of observation. (ii) A boolean flag wait that signals when the process is stable and waiting for interaction with its environment. (iii) A set ref of events that the process is refusing during its wait state. There are three healthiness conditions on these observations, but we concentrate on just one:*

$$\mathbf{R1}(P) \mathrel{\widehat{=}} P \wedge tr \leq tr'$$

This monotonic idempotent function requires the history to be unchanged.

Definition 27 (CSP processes). *CSP processes are reactive processes with two additional healthiness conditions that mirror those for designs; but note the significant difference in the first condition.*

$$\mathbf{CSP1}(P) \mathrel{\widehat{=}} \mathbf{R1}(\neg\, ok) \vee P$$
$$\mathbf{CSP2}(P) \mathrel{\widehat{=}} P \,;\, J$$

Theorem 28 (CSP-design coretraction). *$(\mathbf{H}, \mathbf{CSP} \circ \mathbf{R1})$ is a coretract.*

Proof. *We begin by proving that $\mathbf{CSP} \circ \mathbf{R1} \circ \mathbf{H}(P) = P$, for a CSP process P:*

$$\begin{aligned}
&\mathbf{CSP} \circ \mathbf{R1} \circ \mathbf{H}(P) \\
&= \{\ \textit{definition: } \mathbf{H}\ \} \\
&\mathbf{CSP} \circ \mathbf{R1} \circ \mathbf{H1} \circ \mathbf{H2}(P) \\
&= \{\ \textit{lemma: } (P = \mathbf{R1}(P)) \Rightarrow \mathbf{CSP1}(P) = \mathbf{R1} \circ \mathbf{H1}(P)\ \} \\
&\mathbf{CSP} \circ \mathbf{CSP1} \circ \mathbf{H2}(P) \\
&= \{\ \textit{definition: } \mathbf{CSP2}\ \} \\
&\mathbf{CSP} \circ \mathbf{CSP1} \circ \mathbf{CSP2}(P)
\end{aligned}$$

$= \{$ *assumption:* P *is* **CSP***-healthy* $\}$
P

Next, we prove that $\mathbf{H} \circ \mathbf{CSP} \circ \mathbf{R1}(D) \sqsupseteq D$*:*

$\mathbf{H} \circ \mathbf{CSP} \circ \mathbf{R1}(D)$
$= \{$ *definition:* **CSP1** $\}$
$\mathbf{H2} \circ \mathbf{H1} \circ \mathbf{CSP1} \circ \mathbf{CSP2} \circ \mathbf{R1}(D)$
$= \{$ *lemma:* $\mathbf{H1} \circ \mathbf{CSP1}(P) = \mathbf{H1}(P)$ $\}$
$\mathbf{H2} \circ \mathbf{H1} \circ \mathbf{CSP2} \circ \mathbf{R1}(D)$
$= \{$ *lemma:* **H1–H2** *commute* $\}$
$\mathbf{H1} \circ \mathbf{H2} \circ \mathbf{CSP2} \circ \mathbf{R1}(D)$
$= \{$ *lemma:* $\mathbf{H2} \circ \mathbf{CSP2}(P) = \mathbf{H2}(P)$ $\}$
$\mathbf{H1} \circ \mathbf{H2} \circ \mathbf{R1}(D)$
$\sqsupseteq \{$ *lemma:* **H** *monotonic* $\}$
$\mathbf{H1} \circ \mathbf{H2}(D)$
$= \{$ *assumption:* D *is* **H***-healthy* $\}$
D

5 Conclusions

We have shown how UTP can be used to construct semantic theories for particular programming paradigms. The main example that we presented, designs, is a contractual theory of total correctness for a nondeterministic sequential programming language with an embedded subtheory underpinning separation logic. In Sect. 4, we introduced two further theories for reactive processes and for CSP processes, and showed that CSP is a coretraction of the theory of designs.

The benefit that arises from this embedding of designs in the CSP world is that it imports the assertional reasoning technique from sequential programming into concurrent programming in CSP. Every CSP process can be expressed as a reactively healthy design $\mathbf{R}(P \vdash Q)$. Hoare logic can now be defined in reactive theories as $\{ p \}\ Q\ \{ r \} \ = \ \mathbf{R}(p \vdash r') \sqsubseteq Q$. The standard rules of Hoare logic, augmented perhaps by those for separatin logic, can now be extended to all elements of the signature of the theory of CSP. This includes rules for reasoning about concurrency, nonterminating recursive processes, renaming, hiding, prefixing, input, output, etc.

6 Acknowledgements

The work reported in this paper is partially supported by the European Commission INTO-CPS project (Horizon 2020, 664047).

References

1. Michael J. Banks and Jeremy L. Jacob, On Modelling User Observations in the UTP, in [22]:101–119 2010.
2. Riccardo Bresciani and Andrew Butterfield, A Probabilistic Theory of Designs Based on Distributions, in [27]:105–123 2012.
3. Andrew Butterfield, Saoithín: A Theorem Prover for UTP, in [22]:137–156, 2010.
4. Ana Cavalcanti and Marie-Claude Gaudel, Specification Coverage for Testing in Circus, in [22]:1–45 2010.
5. Andrew Butterfield (ed,), *Unifying Theories of Programming*, Second International Symposium, UTP 2008, Dublin, 8–10 September 2008, Revised Selected Papers, Springer LNCS 5713 2010.
6. Ana Cavalcanti, Alexandre Mota and Jim Woodcock, Simulink Timed Models for Program Verification, Zhiming Liu, Jim Woodcock and Huibiao Zhu (eds), *Theories of Programming and Formal Methods*, Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday, Springer LNCS 8051 82–99 2013.
7. Ana Cavalcanti, Andy J. Wellings and Jim Woodcock, The Safety-critical Java memory model formalised, *Formal Asp. Comput.* 25(1):37–57 2013.
8. Ana Cavalcanti and Jim Woodcock, A Tutorial Introduction to CSP in *Unifying Theories of Programming*, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock (eds), Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, 23 November–5-December 2004, Springer LNCS 3167 220–268, 2004.
9. Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson and Hongseok Yang, Views: compositional reasoning for concurrent programs, Roberto Giacobazzi and Radhia Cousot (eds), 40th Annual ACM Symposium on Principles of Programming Languages, POPL '13, Rome, Italy, 23–25 January 2013, 287–300, 2013.
10. Steve Dunne and Bill Stoddart, *Unifying Theories of Programming*, First International Symposium, UTP 2006, Walworth Castle, County Durham, 5–7 February 2006, Revised Selected Papers, Springer LNCS 4010 2006.
11. Simon Foster and Jim Woodcock, Unifying Theories of Programming in Isabelle, Zhiming Liu, Jim Woodcock and Huibiao Zhu (eds), *Unifying Theories of Programming and Formal Engineering Methods*, International Training School on Software Engineering, Held at *ICTAC 2013*, Shanghai, 26–30 August 2013, Advanced Lectures, Springer LNCS 8050 109–155 2013.
12. Simon Foster, Frank Zeyda and Jim Woodcock, Isabelle/UTP: A Mechanised Theory Engineering Framework, in [18]:21–41, 2014.
13. Will Harwood, Ana Cavalcanti and Jim Woodcock, A Theory of Pointers for the UTP, John S. Fitzgerald, Anne Elisabeth Haxthausen and Hüsnü Yenigün, *Theoretical Aspects of Computing*, ICTAC 2008, 5th International Colloquium, Istanbul, 1–3 September 2008, Springer LNCS 5160 141–155 2008.
14. Ian J. Hayes, Termination of Real-Time Programs: Definitely, Definitely Not, or Maybe, in [10]:141–154. 2006.
15. Jifeng He, A Probabilistic BPEL-Like Language, in [22]:74–100 2010.
16. Jifeng He, Shengchao Qin and Adnan Sherif, Constructing Property-Oriented Models for Verification, in [10]:85–100 2006.
17. C. A. R. Hoare and He Jifeng , *Unifying Theories of Programming*, Prentice Hall 1998.

18. David Naumann (ed.), *Unifying Theories of Programming*, 5th International Symposium—UTP 2014, Singapore, May 13, 2014, Revised Selected Papers, Springer, LNCS, 8963, 2015.
19. Marcel Oliveira, Ana Cavalcanti and Jim Woodcock, Unifying Theories in ProofPower-Z, in [10]:123–140 2006.
20. Marcel Oliveira, Ana Cavalcanti and Jim Woodcock, A UTP semantics for *Circus*, *Formal Asp. Comput.* 21(1–2):3–32 2009.
21. Juan Ignacio Perna and Jim Woodcock, UTP Semantics for Handel-C, in [5]:142–160 2008.
22. Shengchao Qin, *Unifying Theories of Programming*, 3rd International Symposium—UTP 2010, Shanghai, China, 15–16 November 2010, Springer, LNCS, 6445, 2010.
23. Pedro Ribeiro and Ana Cavalcanti, Angelicism in the Theory of Reactive Processes, in [18], 42–61 2014.
24. John C. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures, 17th IEEE Symposium on Logic in Computer Science, LICS 2002, 22–25 July 2002, Copenhagen, Denmark, 55–74, 2002.
25. Thiago L. V. L. Santos, Ana Cavalcanti and Augusto Sampaio, Object-Orientation in the UTP, in [10]:18–37 2006.
26. Adnan Sherif, Jifeng He, Ana Cavalcanti and Augusto Sampaio, A Framework for Specification and Validation of Real-Time Systems Using *Circus* Actions, Zhiming Liu and Keijiro Araki (eds), *Theoretical Aspects of Computing* ICTAC 2004, First International Colloquium, Guiyang, 20–24 September 2004, Revised Selected Papers, Springer LNCS 3407 478–493 2005.
27. Burkhart Wolff, Marie-Claude Gaudel and Abderrahmane Feliachi (eds), *Unifying Theories of Programming*, 4th International Symposium, UTP 2012, Paris, 27–28 August 2012, Revised Selected Papers, Springer LNCS 7681 2013.
28. Jim Woodcock and Ana Cavalcanti, A Tutorial Introduction to Designs in Unifying Theories of Programming, Integrated Formal Methods, 4th International Conference, 4–7 April, 2004, Eerke A. Boiten and John Derrick and Graeme Smith (eds), Springer LNCS 2999:40–66, 2004.
29. Jim Woodcock, The Miracle of Reactive Programming, in [5]:202–217 2008.
30. Jim Woodcock, Engineering UToPiA—Formal Semantics for CML, Cliff B. Jones, Pekka Pihlajasaari and Jun Sun (eds), *FM 2014: Formal Methods*, 19th International Symposium, Singapore, 12–16 May 2014, Springer LNCS 8442 22–41 2014.
31. Jim Woodcock and Victor Bandur, Unifying Theories of Undefinedness in UTP, in [27]:1–22 2012.
32. Frank Zeyda and Ana Cavalcanti, Encoding *Circus* Programs in ProofPowerZ, in [5]:218–237 2008.
33. Naijun Zhan, Eun-Young Kang and Zhiming Liu, Component Publications and Compositions, in [5]:238–257 2008.
34. Huibiao Zhu, Jifeng He, Xiaoqing Peng and Naiyong Jin, Denotational Approach to an Event-Driven System-Level Language, in [5]:258–278 2008.
35. Huibiao Zhu, Peng Liu, Jifeng He and Shengchao Qin, Mechanical Approach to Linking Operational Semantics and Algebraic Semantics for Verilog Using Maude, in [27]:164–185 2012.
36. Huibiao Zhu, Jeff W. Sanders, Jifeng He and Shengchao Qin, Denotational Semantics for a Probabilistic Timed Shared-Variable Language, in [27]:224–247 2012.
37. Huibiao Zhu, Fan Yang and Jifeng He, Generating Denotational Semantics from Algebraic Semantics for Event-Driven System-Level Language, in [22]:286–308 2010.